1978

On Learning How to Predict

Peter J. Denning

Report Number:
79-316

On Learning How To Predict

(A Keynote Speech)

Peter J. Denning

Computer Science Department
Purdue University
West Lafayette, IN 47907

CSD - TR 316

ON LEARNING HOW TO PREDICT

(A Keynote Speech)*

Peter J. Denning

The organizers of this symposium suggested that I am in a unique position to comment on the future of Computer Performance Evaluation. "Why," I asked. "Any one of us can predict the future. The problem is to discover who is right." To avoid the risk of failure , I decided to concentrate my message on a constructive approach to the future: I decided to suggest things that you can do to make the future come out well. Because we Performance Evaluators are artisans of models, whose primary applications are primarily for prediction, I will concentrate on the issue of getting closer agreement between performance models and real computer systems.

My message is this: Think not of building a good model of the world, but of building the world more like a good model. And then experiment with the resulting system.

---

*Presented at the CPEUG Conference, October 15, 1979.

The first part of this message is of course hyperbole. Much of the world is not of our making; we must be content to find models that aid our understanding of nature. But the world of computers is manmade. We have a real opportunity to guide its course. Rather than passively accept the often-futile assignment to model each new operating system that we do not understand, we can actively push for operating systems that conform closely to models we do understand. Such operating systems would be less mysterious and more predictable.

The second part of this message is even more serious. Many excellent ideas have failed to make the passage from the world of research to the world of practice. This is because no one bothered to carry out the final experiment -- comparing the proposed system with existing systems. There is, sadly, a lack of experimentation with large software systems. Experimentation builds physical intuition, which, in many of us, is not well developed. Experimentation greases the channel of technology transfer.

PRECEDENTS IN SOFTWARE ENGINEERING

The idea of tuning a system to a model, rather than a model to a system, at first seems strange. It has not been popular among operating systems designers, many of whom believe the resulting system would be less flexible, or that it would not utilize resources efficiently, or that it would be a bad system. Yet many operating systems have so much flexibility that we do not know how to control them; contemplate, for example, the many tunable parameters of IBM's MVS. Many operating systems exert such poor control over resource contention that unstable operation results when high utilizations are attempted; contemplate, for example, thrashing in virtual memory systems. Many operating systems are so complex that one cannot tell whether they are bad or good; indeed, some of the most understandable systems today are the simplest and most structured -- UNIX and B6700 are examples.

But I do not want to rely on such vague arguments. Let me turn to the world of Software Engineering for precedents to the idea of building software more like models. Consider these developments:

1. Structured Programming. We are advised to refrain from arbitrary program structures, restricting ourselves to simple forms that correspond directly to the standard proof schemata of case enumeration and mathematical induction. The resulting program

corresponds closely to an intuitive (well understood) model of mathematical proof.

2. Typed Languages. We are advised to employ programming languages that force explicit declarations of all data types, and that provide elementary types such as strings and records. In return for these restrictions, we get automatic type checking from our compilers, proper mode conversions, and error checking in subroutine linkages. The resulting programs correspond closely to intuitive models of data.

3. Abstract Data Types. We are advised to employ programming languages which allow the programmer to extend the basic set of data types. Syntactic forms permit the programmer to localize the definition of a class of objects to a single place in the program, to restrict direct access only to procedures (operations) authorized access to the given type of objects, to hide detailed information about the implementation from the user, and to prevent side effects. The resulting programs correspond closely to an intuitive model of programs as a hierarchy of abstract machines that interact only along well defined interfaces.

4. Distributed Processes. We are advised to consider networks of disjoint processes as models of distributed computing. These processes share no memory and

interact only by messages sent over specified channels. The resulting "program" corresponds closely with our intuitive notion of autonomous processors in a dataflow network.

Some skeptics have argued that all these concepts are so constraining that the resulting system will be larger and more complex than a software system programmed in traditional ways. This may be true for small programs, but large systems are different. Brinch Hansen's experience with the Solo Operating System, written in Concurrent Pascal, shows that the structure helps designers avoid unwanted redundancy, leading thereby to a considerably simpler design for the same set of functions.

The four kinds of models noted above share four basic properties:

1. There is an underlying intuitive model that closely resembles the structure of the problem being addressed. (This is sometimes called the "semantic structure.")

2. The end product, a program or set of programs, has a structure that closely resembles the intuitive model. (This is sometimes called the "syntactic structure.")

3. The end product is highly modular. Each of the components is independent of the others, except to the extent of its interactions via clearly defined

interfaces. Information flow across the interfaces is carefully controlled.

4. The modules form a hierarchy. Each can be refined into smaller modules, and each is the component of a larger module.

When I suggest building systems to conform to a model, I have in mind precisely the kind of activity already practiced by software engineers. We begin with a good semantic model, an intuitive model that deals with the problem in a natural way. We build the system so that its physical structure closely mirrors the abstract structure of the model. We rely heavily on modularity, so that the separate components of the system have much stronger interactions internally than between them, and so that all interactions between components are explicit.

If we follow these patterns, we will construct systems whose behavior is understandable. Instabilities can be eliminated because interactions between components are all explicit and subject to control. A correction to the model, designed to improve the performance of the system, can be implemented easily because semantic changes have immediate interpretations in the physical structure of the system.

Finding intuitive models for the structure of software systems is not enough. We also need convincing evidence that the result is better than the software that came

before. Experience has shown that the model of proof underlying structured programming and the model of data underlying typed languages are good models. Experience with abstract data types is limited (I know of only four working compilers in the U.S.); Brinch Hansen's Solo Operating System, which is expressed in 1300 lines of Concurrent Pascal, illustrates the simplifying power of the technique. Experimentally tested models of distributed processes have yet to be designed.

To illustrate these ideas, I will consider four examples. In each case, a model was used to structure a system in order to elicit the desired behavior, and experiments were performed to test the results. The models are: IBM's TSO (Time Sharing Option), IBM's MVS (Multiple Virtual Storage for the 370 series), optimal multiprogrammed memory management, and queueing network models of computer systems. I will comment on the role played by experiments in bringing each of these models into practice.

Example 1 -- TSO

After he showed in 1965 that the machine repairman model of queueing theory was an excellent model of CTSS at MIT, Allan Scherr went on to IBM, where he was given the unenviable task of devising a performance model for TSO. It

did not take long for him to conclude that TSO was intractable, so he set about modifying TSO to fit the machine repairman model. This was duly done. Accurate forecasts of TSO throughput and response time were then possible. Scherr's insight was significant.

A UNIX enthusiast is reputed to have said that using TSO is like kicking a dead whale along the beach. This is not the fault of Scherr's model. Scherr was asked to rescue TSO, not to design a better time sharing system. At least it can be said that the reasons for TSO's poor performance were well understood. Major improvements, which would have required fundamental changes to the architecture of the 360 and to O.S., were out of the question.

Example 2 -- MVS

Nearly a dozen years ago, I suggested the concept of partitioning the workload into classes whose jobs had similar demands, and guaranteeing a portion of the processing and memory hardware to each class. I was attracted to this idea because one could then specify how much hardware would be needed in each class to meet the performance guarantees. (Performance guarantees take the form of throughput minima or response time maxima for each class.) This idea was not taken seriously because it appeared wasteful -- idle

resources in one class could not be reallocated to work off a backlog in another class, and a considerable amount of online instrumentation would be needed to be sure each class received its guaranteed resources.

The designers of MVS rediscovered part of this idea. MVS permits users to define "performance groups" and to specify minimum resources to be associated with each performance group. MVS continually monitors its own performance for conformity to the performance objectives in effect at each time. The designers of MVS did not consider it wasteful, in terms of the potential benefits, to reserve resources and to employ a lot of instrumentation.

Aside from the concept of partitioning the system's resources among job classes, MVS is not based on a previously understood model of a system. It provides numerous parameters for each job class. A group at IBM Research in Yorktown, NY, is busily seeking a model of MVS which, given the settings of the parameters, would calculate performance measures for each job class. Performance consultants are frequently asked to "tune" MVS by generating values for all parameters from a smaller set of independent parameters.

Example 3 -- Multiprogrammed Memory Management

Around 1967, during the middle stages of the design of Multics, Jerry Saltzer characterized the ultimate objctive of the resource allocator in a multiprogrammed, virtual memory system. He said that the problem is to find a completely automatic mechanism with at most one external parameter that can be tuned once to bring the system's throughput near optimum for a wide range of workloads. Since that time several hundred researchers have addressed various aspects of the problem, ranging from analysis of paging algorithms and program behavior, across measurements on programs and experiments with real systems, to queueing network studies of optimal feedback controls. The collective results of all this research have produced impressive evidence that the so-called "working set dispatcher" would be a solution of Saltzer's Problem. A working set dispatcher has three main components:

1. Processes will be either active or inactive. Only the active ones are allowed to hold space in main memory or use processing elements.

2. Simple hardware will measure the working set of each active process by determining the backward reference intervals, in virtual time, of each page loaded in the main store. The measurement of a process's working set will not be influenced by any other processes in the system. One global value of the working set

parameter (the window) suffices to cause all the space-times of active processes to be near minimum, which thereby maximizes system throughput.

3. The highest priority active inactive process will be activated only when there is space in memory sufficient to hold its working set. This is an implicit form of load control.

I know of two experimental dispatchers constructed along these lines. Even without hardware support for working set detection, these systems performed significantly better with the working set dispatcher in operation. One was the CP-67 system at Grenoble, France, and the other was on the Edinburgh Multi Access System, in Scotland. As was demonstrated on the MANIAC II machine at Los Alamos in 1971, the hardware required is cheap -- about a dozen flipflops costing $20 per page frame at that time. Had such hardware been available to the designers of the working set dispatchers for the Grenoble and Edinburgh systems, even more significant improvements would have been observed.

The area of multiprogrammed memory management has, therefore, exhibited both model development and experimental testing of hypotheses. Paradoxically, all this knowledge remains largely unused in real or planned operating systems. Most modern operating systems use the memory policy called CLOCK, a fifteen-year-old design based on a FIFO list of all pages and on usage bits. Because this policy does not

distinguish pages among programs, it cannot guarantee each program a minimum space-time resident set and it cannot avoid thrashing without an auxiliary load controller. The main argument for CLOCK is its simple implementation. Yet, experimental evidence clearly shows that a well tuned CLOCK is still significantly worse than a poorly tuned working set dispatcher. (This includes all overhead.) Because interactions among processes are not controlled by CLOCK -- that is, the principle of modularity is not enforced -- it has proved quite difficult to model CLOCK accurately.

One can understand that modifying a fifteen-year-old operating system to incorporate five-year-old results may not be cost effective. But one has difficulty understanding the VAX DEC's advanced virtual memory machine, which has no usage bits on its page frames! One has difficulty understanding why new operating systems or memory technologies do not attempt to employ these results.

I am not sure how to reconcile the existence of a well-developed, tested model for optimal multiprogrammed memory management with the lack of operating systems that employ the model. There are many pieces in the puzzle -- one has to examine a large body of evidence to see why the working set dispatcher solves Saltzer's Problem. Perhaps few are aware of it.

But I speculate that the major difficulty lies in the widespread belief that advanced memory technology will soon

be so cheap that automatic feedback controls for memory management will be worn out relics. Those who believe this feel little need to build working set dispatchers. They feel little need for hardware to support memory management.

The problem with this belief is that it is wrong. One of the first things I was told about computers in 1954 was that virtual memory would not live long because desk-top computers with ample memory and cheap processing would soon arrive. The predicted memory technology arrived on schedule -- but virtual memory is still here. There are two reasons. First, it is in human nature to push frontiers, applying computers to ever larger problems beyond the capacity of the available main memory. Second, multilevel memories will always be more cost effective than monolithic memories. (This is because a random access memory, which provides a separate access path to each memory location, is always more expensive than a circulating store, which needs just one access path.) The need for reliable long term storage systems and data management systems is shifting the problem of sharing from multiprogramming the main store to multiprogramming the secondary store. The dimensions and scales of the problem change, but not its basic concepts.

The need for the concepts of virtual memory will be with us for a long time to come, and so will the need for the concepts of the working set dispatcher.

Example 4 -- Queueing Network Models

Queueing network models have become immensely popular. I think this is less because they work, than because they are intuitive. Not only do they embody the natural behavioral concepts of systems -- concepts such as job flows, backlogs, and response times -- but their computational algorithms are extraordinarily efficient. So strong is the intuitive appear of queueing network models that they would be used even if we did not know that operational assumptions can replace the stochastic in their derivations.

Queueing network models are also attractive because they strongly embody the property of modularity I noted earlier -- in this case called "decomposability". A subsystem can be replaced with a single service station whose output rate under a given instantaneous load is the same as the long-term throughput of the subsystem under a constant load. This replacement will be nearly exact if the subsystem interacts weakly with other subsystems. Using decomposition, we can reduce complex systems to simple networks whose behavior is easily understood.

Queueing network models give excellent results when the systems they model are constituted of nearly-decomposable subsystems. A useful rule of thumb is that a subsystem, or a service station, is perfectly decomposable when the coefficient of variation of times between job completions is exactly 1, and nearly decomposable when this coefficient is

less than 2. (The coefficient of variation is the ratio of the standard deviation to the mean; it measures the skewness of the distribution.)

Researchers have been seeking approximations for the common case of systems containing indecomposable service stations. Not surprisingly, the most successful approxmations are derived from decomposable queueing network models. Starting with a queueing network model whose parameters are the mean total service requirements at the network's stations, these methods iteratively alter the model's parameters to incorporate the empirical coefficients of variation of service. The best of these methods converge on a decomposable model that approximates the real system's throughputs to within 10% and mean queue lengths to within 25%, for coefficients of variation all the way up to 10. This may be sufficient for most practical cases.

Queueing network models also show why a station of high coefficient of variation is indecomposable: the occasional very long blocks the majority of jobs, which are very short in comparison. A large dispersion in job sizes generates backlogs not accounted for in decomposable models. Queueing network models also tell us that stations with multiple internal paths, by which subsequently arriving short jos can bypass the occasional long one, are nearly decomposable. In other words, serialism is the enemy of decomposability, while parallelism is its friend.

The familiar round robin schedule is a practical way to approximate a "processor sharing" station, which processes all waiting jobs in parallel. It has been used for years in time sharing systems to prevent long jobs from blocking short jobs by monopolizing the processor. The central server queueing network, which is a good model of the active part of most operating systems, has enough internal parallelism so that under real operating conditions, its output process has low coefficient of variation. (We have observed, for example, a coefficient of variation of 1.05 in our system at Purdue, indicating nearly perfect decomposability.) This means that the "central subsystem" is decomposable and can be accurately replaced by a single station in a high level model of the system, irrespective of the details of CPU and I/O usage within the subsystem.

These examples suggest a possible way of tuning a system so that it becomes more like a good model: indecomposable subsystems can be modified by introducing more internal parallelism. The resulting system will behave more like an intuitive model. It will easier to understand and predict.

CONCLUSIONS

In computer science, we use models quantitatively and qualitatively. Quantitatively, the model gives a set of

equations by which we can calculate the values of certain variables from measurements of others. Qualitatively, the model helps us better understand the relations among the parts of the system -- to bring complexity under control.

We are accustomed to thinking of performance models quantitatively and software models qualitatively. But many of our analytic performance models have the very properties that make software models valuable: an intuitive basis natural to the problem at hand, a physical structure that closely resembels the intuitive basis, and hierarchical modularity. I am suggesting, therefore, that we try to apply performance models in the design of large software systems. The result would be software systems whose behavior is not only understandable, but predictable.

You must apply this advice wisely. A system that conforms to a well understood model may be understandable without having acceptable performance, as illustrated by TSO. A system may set out to conform to a model so general that its many independent parameters confound understanding and complicate tuning, as illustrated by MVS.

In following this advice, you may be led to proposals for improving the design of a system. But if you want your proposal accepted, you must back it up with experimental facts. Those who implement systems are notoriously conservative. Unless your case is supported by data, they will not pay you much heed. You must experiment to succeed.

The need for experimenting with proposals to discover whether they are meritorious is not widely recognized in the field. Experimental facilities in both universities and industry are generally inadequate. Without facilities, and without a commitment to testing, there will be no experimenting. Without experimenting, there will be no technology transfer, no passage of ideas from the world of research to the world of practice.

Adequate experimenting does not ensure that an idea will be accepted into practice. Although the solution of the memory management problem is well supported by data and has been successfully implemented in at least two experimental systems, its transfer into practice has been blocked by the naive notion that cheap memory will save us from our current problems. This notion, born of a lack of physical intuition as would be fostered by experimentation, holds that cheap microprocessors and new memory technology will soon obviate all that we have learned about protecting objects stored in a shared memory systems. This notion completely misunderstands the nature of the problems solved by our approaches to memory.

So go and study ways of tuning systems to fit models. Experiment with your results. And do not think hardware technology is going to solve your software problems.